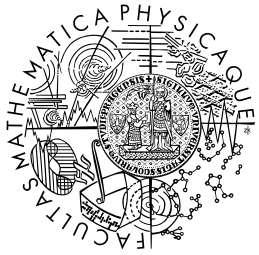


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Marián Kažimír

Using Constrained Horn Clauses for Hierarchical Planning

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Martin Blicha
Study programme: Computer Science (B1801)
Study branch: Programming and SW systems

Prague 2021

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to kindly thank my supervisor Mgr. Martin Blicha and my consultant RNDr. Jan Kofroň, Ph.D. for their neverending patience, help and leadership that have brought me to this place and to all who supported me on this long journey!

Title: Using Constrained Horn Clauses for Hierarchical Planning

Author: Marián Kažimír

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Blicha, Department of Distributed and Dependable System

Abstract: HTN planning is a popular approach to modelling planning problems. The goal of this project is to implement a HTN solver based on translating the planning problem to a reachability problem in a transition system and using off-the-shelf reachability solver for the actual solving.

Keywords: Hierarchical Task Networks Planning problem Solver Transition system Constrained Horn Clauses

Název práce: Užití omezených Hornových klauzulí pro hierarchické plánování

Autor: Marián Kažimír

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Martin Blicha, Katedra distribuovaných a spolehlivých systémů

Abstrakt: HTN plánování je populární způsob jak modelovat plánovací problémy. Cílem této práce je naimplementovat HTN řešič založený na překladu plánovacího problému do problému dosažitelnosti v tranzitním systému a ten následně vyřešit pomocí běžného řešiče dosažitelnosti.

Klíčová slova: HTN Plánovací problém řešič Tranzitní systém Omezené Hornove klauzule

Contents

Introduction

Technologies have become undoubtedly an inseparable part of our lives. More and more people rely on them and are addicted to them, and time spent near them is increasing rapidly. What's more, we teach machines to be independent and operate without any control. Until recent times, behind every decision was a human, however now – autonomous cars, industry robots, drones, etc., these all make decisions that can affect people. Humans can think, machines cannot, so how can I know my vehicle will find me? Or can the drone bring me my shipment? We could ask similar questions forever, but the answer is planning!

According to Wikipedia, "automated planning and scheduling, sometimes denoted as simply AI planning, is a branch of artificial intelligence that concerns the realization of strategies or action sequences, typically for execution by intelligent agents, autonomous robots, and unmanned vehicles"[1]. But how can one say whether the task is doable or even come up with a plan for such a task? Currently, incremental SAT (Boolean satisfiability problem) solvers are most commonly used for planning and scheduling task problems. When applying SAT solving to these problems, the whole procedure features four steps: (1) enumerating and instantiating all the possible actions, (2) encoding the instantiated problem into propositional logic, (3) finding a solution with a SAT solver, (4) decoding the found variable assignment back to a valid plan. This SAT-based approach generally shows the best performance.

The disadvantage of this method is that one needs to build and issue a sequence of queries representing plans of fixed lengths. If no plan is found, the same is repeated for incremented length and so on. This is something we can overlook for small domains, but in tasks that grow to enormous size, the overhead becomes a non-trivial component. There comes a question if there is a better, faster, and easier way to do this. The aim of this work was to investigate a different approach to solve planning and scheduling tasks using a transition system.

In the beginning, for a given domain and problem, all possible states will be modeled in the planning problem and transformed into a transition system based on Constrained Horn Clauses (CHC) format. Then the existence of a plan is equivalent to the reachability of the goal state in this transition system. To make things easier, the focus and realization of this thesis will cover only totally-ordered Hierarchical Task Networks (HTN) planning problems.

In the first chapter, we introduce the basic knowledge needed to understand this topic. The second chapter contains information on how the whole program holds together with its main parts and functions. The following chapters provide details about our approach, algorithms, technology stack, test data. In the end, we present several benchmarks, comparisons with the native method, and findings that were acquired during development.

1. Basic knowledge

In this section we present definitions that were mostly obtained from paper HDDL – A Language to Describe Hierarchical Planning Problems [2]

1.1 Hierarchical Task Networks (HTN) planning

Hierarchical planning, or Hierarchical Task Networks planning in full, uses the language of first-order logic with the following sets: T is a set of type symbols, V is a set of typed variable symbols, P is a set of predicate symbols, and C is a set of typed constants. There are two distinct kinds of tasks: primitive tasks (also called **actions**) and **compound tasks** (also called abstract tasks).

Definition 2: A task network tn over a set of task names X (first-order atoms) is a tuple $(I, \prec, \alpha, VC)$ with the following elements:

1. I is a (possibly empty) set of task identifiers.
2. $\prec$ is a strict partial order over I.
3. $\alpha : I \to X$ maps task identifiers to task names.
4. VC is a set of variable constraints.

Each constraint can bind two task parameters to be (non-)equal, and it can constrain a task parameter to be (non-)equal to a constant or to (not) be of a certain type. A task network is ground if all parameters are bound to (or replaced by) constants from C.

An **action** a is a tuple $(name, pre, \mathit{eff})$, where $name$ is its task name, a first-order atom consisting of the (actual) name followed by a list of typed parameter variables. pre is its **precondition**, a first-order formula over literals over predicates from P. eff is its **effect**, a conjunction of literals over predicates from P. We define eff^+ and eff^- as the sets of atoms occurring non-negated/negated in eff. All variables used in pre and eff have to be parameters of $name$. We also write $name(a)$, $pre(a)$, and $\mathit{eff}(a)$ to refer to these elements. We require that action names $name(a)$ are unique.

A **compound task** is simply a task name, i.e., an atom. Its purpose is not to induce state transition, but to reference a pre-defined mapping to one or more task networks by which that compound task can be refined. This mapping is given by a set of (decomposition) methods M. A method $m \in M$ is a triple (c, tn, VC) of a compound task name c, a task network tn, and a set of variable constraints VC that allow to (co)designate parameters of c and tn.

1.1.1 Planning Domain (PD)

Definition 3: A planning domain D is a tuple (L, TP, TC, M) defined as follows:

- L is the underlying predicate logic.
- TP and TC are sets of primitive and compound tasks.
- M is a set of decomposition methods with compound tasks from TC and task networks over the names $TP \cup T$.

The domain implicitly defines the set of all states S, being defined over all subsets of all ground predicates.

Additionally, a domain is **totally-ordered** iff the subtasks in all methods and in the initial task network form a sequence, i.e. the declared ordering arranges the tasks in a sequence.

1.1.2 Planning Problem (PP)

Definition 4: A planning problem P is a tuple (D, sI, tnI, g), where:

- D is a planning domain.
- $sI \in S$ is the initial state, a ground conjunction of positive literals over the predicates.
- tnI is the initial task network (not necessarily ground).
- g is the goal description, a first-order formula over the predicates (not necessarily ground)

1.1.3 Hierarchical Domain Definition Language (HDDL)

Hierarchical Domain Definition Language (HDDL) is a common input language for hierarchical planning problems. This file format was proposed in the 2019 and was used for the first International Planning Competition for Hierarchical Planning. The complete grammar can be found in the paper HDDL – A Language to Describe Hierarchical Planning Problems [2]. We will analyze one example of PD and PP in the HDDL language in the Section 3.

1.2 Constrained Horn Clauses (CHC)

Definition 1: A Constrained Horn Clause is a formula

$$C \ \wedge \ P_1 \ \wedge \ \ldots \ \wedge \ P_n \ \implies \ H \tag{1.1}$$

where

- C is a constraint over the background theory.
- $\forall\ P_i \in \{\ P_1 \dots P_n\ \}$ is an uninterpreted predicate applied to (interpreted) terms over the background theory.
- H is an uninterpreted predicate applied to interpreted terms, or it is the constant $\bot$.

H is called the head of the clause and $C \wedge P_1 \wedge \ldots \wedge P_n$ is called the body. Equivalently, we can write CHC in a Prolog format

$$H\ :-\ C\ \wedge\ P_1\ \wedge\ \ldots\ \wedge\ P_n \tag{1.2}$$

A set of CHCs is (syntactically) solvable if there exists an interpretation of the uninterpreted predicates in the background theory such that each CHC yields a valid formula when the uninterpreted predicates are substituted by their interpretation. If there exists a *ground* derivation of the constant *false* ($\bot$) then the system of CHCs is not solvable.

Example 1. Consider the following system of CHCs with the background theory of integer linear arithmetic.

$$\begin{aligned} x = 0 \wedge y = 0 &\implies Inv(x, y) \\ Inv(x, y) \wedge x' = x + y \wedge y' = y + 1 &\implies Inv(x', y') \\ Inv(x, y) \wedge x < 0 &\implies \bot \end{aligned} \tag{1.3}$$

The solution for this set of CHCs it is the interpretation $x \geq 0 \wedge y \geq 0$ for $Inv(x, y)$.

Example 2. Consider this system of CHCs with the background theory of integer linear arithmetic.

$$\begin{aligned} x \leq 0 &\implies P(x) \\ P(x) \wedge x < 5 \wedge x' = x + 1 &\implies P(x') \\ P(x) \wedge x \geq 1 &\implies \bot \end{aligned} \tag{1.4}$$

In this example, no interpretation that would make all the clauses valid exists. See the proof tree:

$$\dfrac{\dfrac{}{x \mapsto 1}\ \dfrac{\dfrac{\dfrac{}{x \mapsto 0}\ \dfrac{}{x \leq 0 \implies P(x)}}{P(0)}\quad \dfrac{}{P(x) \wedge x < 5 \wedge x' = x + 1 \implies P(x')}\ x \mapsto 0, x' \mapsto 1}{P(0) \implies P(1)}\quad \dfrac{}{P(x) \wedge x \geq 1 \implies \bot}}{P(1) \implies \bot}}{\bot}$$

CHCs have gained popularity especially for their usage in automatic program verification [3]. More details about CHC can be found in paper The Science, Art, and Magic of Constrained Horn Clauses [4]. We introduce our encoding into the CHC clauses in the Section 4.1.

2. Analysis and architecture

2.1 Motivation

The beginnings of Hierarchical Task Network date back to the age of approximately 45 years ago, however they have gained popularity in recent years. Over time, they proved to be helpful in a wide range of application not only because they cope with various properties of an application in the real world. The first successes came from the Simple Hierarchical Ordered Planner (SHOP) [5], and its successors, but they require well-written and almost algorithmic-like domain knowledge. This fact contributed to SHOP being disqualified from the International Planning Competition (IPC) in 2000 because the domain knowledge was not well written, thus the output plan was not valid. With the growing applicability of AI and its development, there is a high demand for the ability to solve such problems quickly and easily. Currently, the most popular approach is to use incremental SAT solvers. While conventional SAT solving processes a single formula in an isolated manner, incremental SAT solving allows for multiple solving steps while successively modifying the formula. However, this obviously includes numerous alterations during the calculation phase.

Our aim was to find a way to avoid complex and time-consuming changes in formulas during the solving phase. But how to do it? We needed to encode the problem into the set of once-created statements. One set of statements encoded every information required - domain, problem, initial state, and goal. Finally, we came with a solution how to achieve this in an elegant way.

We could encode our domain facts and predicates to simple boolean formulas, but how can we represent the objects? Constrained Horn Clauses provide everything necessary to cope with these needs successfully. Once the domain, problem, initial state, and goal are encoded into CHC formulas, this set of clauses is passed to the CHC solver, which provides an answer, whether the goal is reachable or not. Moreover, should the goal be reachable, we can compose the plan from the solver's answer. This approach looked promising, so we designed the encoding and looked at the technical aspects in order to test it.

2.2 Technical analysis

Having our idea in mind, we needed to find the right way to do it. To be able to solve a problem, few steps are required. For IPC, the input consists of two files - domain and problem. Before we could even start encoding, we had to parse these files into the form when later work is comfortable and straightforward. Once the input is parsed, some amendments are needed, and at the end comes the encoding itself.

From the variety of programming languages, the ultimate choice was Java. Java was designed to be easy to use and easy to write, compile, debug, and learn than other programming languages. Some may protest that C++ would be a better choice due to its speed. Java allows us to test our approach more quickly and easily.

From the minimum of HDDL parsers we found, none was suitable for our needs. We needed a parser that would convert input into the appropriate inner representation so we could convert it into the CHC clauses. What's more, we decided to put a strong emphasis on easy integration with other parts of our solver and the ability to make amends if needed. For that reason, the decision was to create our own parser, which would fulfill all these mentioned requirements.

We could build our parser from scratch, yet as it is true that programmers are great at reinventing the wheel again and again, we wanted to come up with a more sublime approach. What's more, such process would be unnecessarily time-consuming, error-prone, and harder to perfect.

In the end, we decided to create a parser using JavaCC (Java Compiler Compiler) for multiple reasons. JavaCC is the most popular parser generator for use with Java applications. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar [6]. As our whole program is written in Java, the decision to use precisely this one from various offers of parser generators was simple. Not only we just had to write a grammar of HDDL, but also follow-up integration with other parts was easy. The grammar itself in *.jj file format is present in our program repository, thus in case of need, it can be amended and parser regenerated.

Each logical unit of HDDL language has its own class. Both PD and PP have fields for all possible data to be stored, and values are set during parsing. After parsing is finished, the internal representation is ready to be amended for further use.

What about CHC? Once the set of clauses is prepared, this set must be given to some solver that supports CHC. The decision, in this case, was to use Z3 theorem prover [7]. The motivation behind using Z3 is simple. There are not many such solvers and Z3 provides a module called Spacer [8] that is currently the best CHC solver (see Competition Report: CHC-COMP-20 [9]). Furthermore, it supports Java API and it has a very useful documentation [10], so it was the best fit.

We were also looking at how to present the outcome of the problem to the user. The standard output used in IPC would be excellent, but for the beginning, there was a need to visualize the traversal of the solver. Here, we chose to form a graph in DOT format as it can be printed to the console, easily stored, and visualized. Once we understood the output structure, conversion into the standard IPC output was straightforward.

Overall, these were the main milestones in the architecture phase where serious decisions were needed. Adding a minor addition as enrichment phase, where we adjust the parsed data, we get the main components and architecture diagram in the next section.

2.3 Main Components

The primary function of this software work, as the name states, is to solve hierarchical planning problems using the Constrained Horn Clauses. Nevertheless, as mentioned above, it is necessary to implement some auxiliary parts as well. How do we get the input? For this purpose, we had to add parsing of the in-

put files. Once the files are parsed, we create the internal representation of the given PD and PP in a suitable structure, then translate PD and PP from internal representation into a transition system. Finally, the problem is solved using our transition system-based format given to the Z3 theorem prover. Z3 then returns the answer in its format, and there is a need to extract the plan from it. Should the answer be negative and the solution does not exist, we print the negative message. In another case, we print the plan. This whole process can be sum up into the simple flow diagram in Figure 2.1.

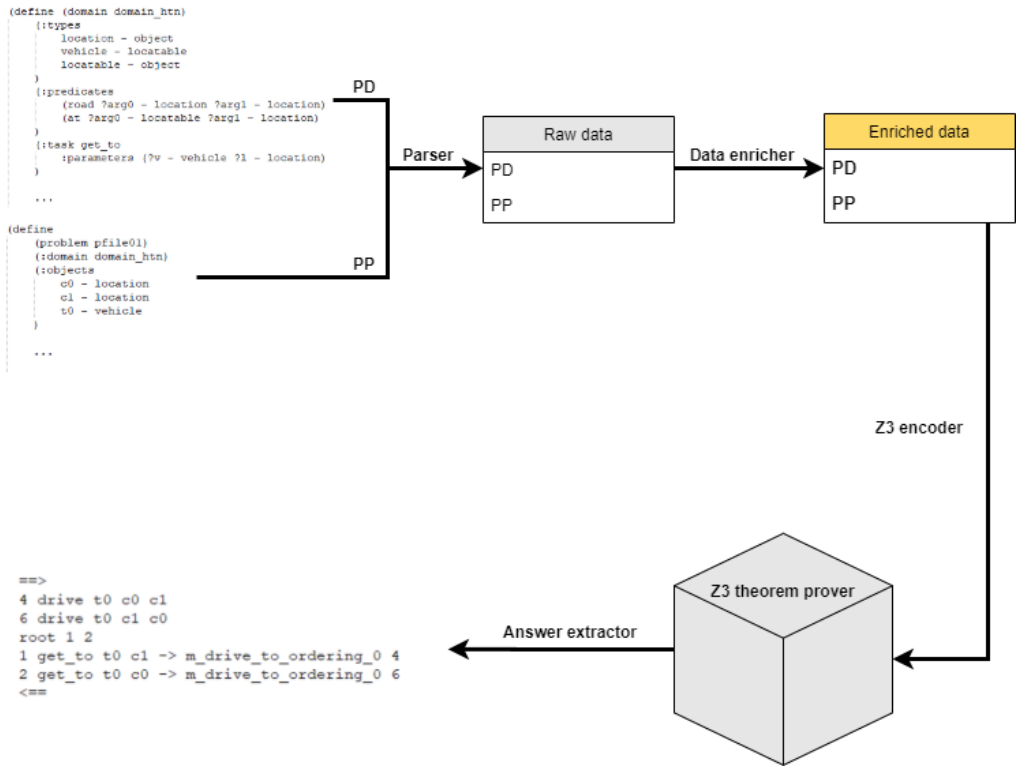


Figure 2.1: Flow diagram of our approach

3. Input and output

3.1 Input format

Totally-ordered HDDL (1.1.3) was chosen as our main input format because it is the common format within this field. An input is totally-ordered iff the subtasks in all methods, and the initial task network form a sequence. The program takes two parameters - PD (1.1.1) and PP (1.1.2). In the chapter 3.2 and 3.3 below is shown an example of such files with a fundamental overview. For more detailed grammar and syntax definitions, we recommend checking HDDL – A Language to Describe Hierarchical Planning Problems [2] paper. As testing data, example tasks from the IPC web page [11] were used.

3.2 PD input format example

domain_htn:

```
(define (domain domain_htn) % domain name definition

    (:types                 % types definitions - all types that can be later
        location - object   % used as parameters in predicates and tasks
        vehicle - locatable
        locatable - object
    )

    (:predicates            % predicates definitions including their signature
        (road ?arg0 - location ?arg1 - location)
        (at ?arg0 - locatable ?arg1 - location)
    )

    (:task get_to           % abstract task signature definition
        :parameters (?v - vehicle ?l - location)
    )

    (:method m_drive_to_ordering_0
        :parameters (?l1 - location ?l2 - location ?v - vehicle)
        :task (get_to ?v ?l2)           % single method decomposing get_to
        :subtasks (                     % abstract task above to subtasks
            task0 (drive ?v ?l1 ?l2)
        )
    )

    (:method m_drive_to_via_ordering_0
        :parameters (?l2 - location ?l3 - location ?v - vehicle)
        :task (get_to ?v ?l3)           % single method decomposing get_to
        :subtasks (and                  % abstract task to subtasks
            (task0 (get_to ?v ?l2))
            (task1 (drive ?v ?l2 ?l3))
        )
    )

    (:action drive
        :parameters (?v - vehicle ?l1 - location ?l2 - location)
        :precondition           % primitive task definition including
            (and                % signature, preconditions (predicates) which
            (at ?v ?l1)         % must be true in order to be able to execute
            (road ?l1 ?l2)      % this task and effects (predicates) which are
        )                       % true as a result of this task being executed
        :effect
            (and
            (not (at ?v ?l1))
            (at ?v ?l2)
```

```
47          )
48      )
49  )
```

This is a trivial example of PD definition in HDDL format. Let's analyze what each section means. The first line of `domain_htn` file holds the name of PD. The next block called *types* defines all types that can be present in this domain – there can be objects of type *location*, *vehicle*, and *locatable*. The names of types are completely arbitrary and do not need to comply with any strict rules. Their purpose is just to restrict the usage of objects in predicates to proper types so we do not end up with something inappropriate like cities being moved and other such cases we would like to avoid.

The next are definitions of abstract tasks. In this case, we have only one abstract task *get_to*. This section called *task* defines the signature of the abstract task, which can later have multiple concrete implementations, which are called methods. We declared that *get_to* is used with the first parameter of type *vehicle* and the second of type *location*.

Afterward come methods – concrete implementations of abstract tasks declared above. This is done via sections called *method*. We have here two implementations of the same abstract task *get_to*. One starting at line 19 called *m_drive_to_ordering_0* with three local variables in section *parameters*. Their usage is limited to be used within this *method* block only. The section *task* here provides pairing to the parent abstract task definition with the right local variables. Lastly, in this block, the section *subtasks* holds the whole abstract task's ordered decomposition to single smaller tasks. Execution of the abstract task means the execution of all its subtasks in the given order. The second implementation of abstract task *get_to* at line 27 called *m_drive_to_via_ordering_0* is similar where the main difference is that it has two subtasks – *get_to* and *drive*.

Finally, we declare all primitive tasks using blocks *action* – in our case, it is only one called *drive*. As in the abstract tasks section, we provide the signature via *parameters* block. Then we state the limitations for executing this task in the block *precondition*. All the predicates present there must be true for this task being executed, otherwise this task in the current state cannot be performed. Also, the block *precondition* can be empty, which would result in the option of executing this task in whatever state. If all preconditions are fulfilled and the task is performed, section *effect* describes the impacts of this task. All predicates there are true once the task is completed, and other predicates, whether preconditions or the ones not being there, have the same value as before, so they are not affected. In the light of this information, action *drive* takes three parameters – vehicle v and locations $l1$ and $l2$. Should we want to execute this task, the vehicle $v1$ must be at the location $l1$, and there must be a road from the location $l1$ to the location $l2$. Supposing, we executed this task, section *effect* changes current state – vehicle v is not at the location $l1$ anymore but at the location $l2$.

3.3 PP input format example

pfile01:

```
1  (define (problem pfile01)   % problem name definition
```

```
2
3       (:domain domain_htn)    % name of domain into which this problem belongs
4
5       (:objects               % concrete objects and their types definitions
6           c0 - location
7           c1 - location
8           t0 - vehicle
9       )
10
11      (:htn
12          :subtasks (         % definition of the goal task
13              task0 (get_to t0 c1)
14              task0 (get_to t0 c0)
15          )
16      )
17
18      (:init                  % initial state definition - predicates
19          (road c0 c1)        % explicitly listed here are true,
20          (road c1 c0)        % all the others are implicitly false
21          (at t0 c0)
22      )
23  )
```

The `pfile01` file is a trivial example of PP within the `domain_htn` domain that can serve as a great example. The first line holds the name of problem in the same manner as we had in the domain file.

Next, at the line 3 is a definition to which domain this problem belongs. Let's say we have multiple domains loaded in the system. Thanks to this section, we know to which domain the problem should be paired to.

The block *objects* declares all objects used within this problem and their types. Here we have two objects of the type *location* called co and $c1$, and one object $t0$ of the type *vehicle*.

Here follows the block *htn* where are the tasks to be achieved. It is the heart of the problem file where the goal is stated. The goal of this example problem is to get to the city $c1$ and back to the city $c0$ using the vehicle $t0$.

Lastly, we have a block called *init* where is the declaration of the initial state. All predicates explicitly listed there are true, and all the others are implicitly false. This means the vehicle $t0$ is at the location $c0$, and there is a road from the $c0$ to the $c1$ and vice versa. We can imagine the whole problem using a simple diagram.

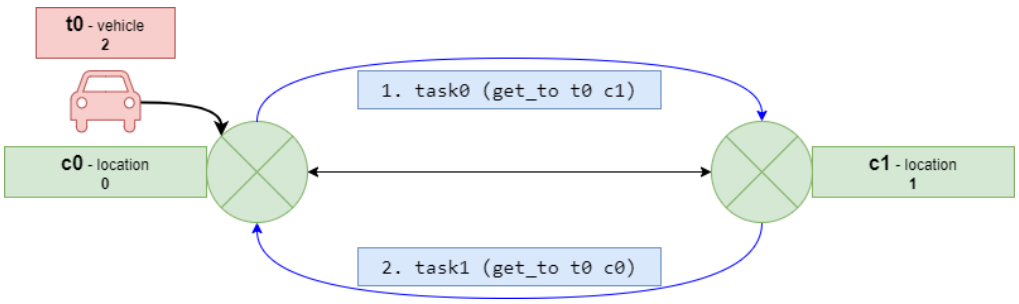


Figure 3.1: $pfile01$ simple description diagram

3.4 Inner representation

Once the PD and PP are parsed, we have all the information from the input files stored in the program in the form of two objects - one PD and one PP. All logical units have their own classes where the data are stored. The structure of data is equal to the structure which is used in input files. The class diagram in Figure 5.1 from the Developer documentation Section 5.1 shows the internal representation of input data with few added elements that are used in the following steps.

Because we wanted to correctly separate the parser and later processing, right after the parsing ends, we have just raw data. It means that some important information is missing. For example, the unique integer ID of each object. This and everything else needed for encoding is added in a phase we call the enrichment phase that is described in the Section 4.3.

3.5 Output format

Our solver uses the official IPC output format that is supported by the plan validator utilized in the competition[12]. It gives the opportunity to easily check the validity of the plan, which comes in handy, especially in the case of dimensional domains and problems where such a check would be burdensome. In short, firstly, we list all primitive actions with unique ID and parameters in the order of execution, then we list all the abstract tasks with a unique ID, concrete method names that were used to fulfill the task and parameters. To illustrate, below is the output in standard form for our example combination of PD (3.2) and PP (3.3) from previous chapters.

Output example:

```
1 ==>
2 4 drive t0 c0 c1
3 6 drive t0 c1 c0
4 root 1 2
5 1 get_to t0 c1 -> m_drive_to_ordering_0 4
6 2 get_to t0 c0 -> m_drive_to_ordering_0 6
7 <==
```

The output starts with "==>" sequence followed by primitive tasks (actions), then there is a line starting with "root" that separates primitive tasks and abstract tasks followed by abstract tasks that are ended by enclosing sequence "<==". Numbers at the beginning do not have to be strict – they are used for referencing between tasks.

In our example, the solver found a plan, so the problem does have a solution. The goal can be achieved by firstly driving a vehicle $t0$ from location $c0$ to the location $c1$ and then driving a vehicle $t0$ from location $c1$ to the location $c0$. These primitive tasks are done as a part of abstract task *get_to* using the method *m_drive_to_ordering_0*

What's more, during development, we implemented a function to visualize the solution in the form of a DOT graph. As this feature can help in numerous cases, we left the ability to get this type of output as well. The tutorial for this feature and everything else needed to run is described in the user documentation (6.2). In case we use "-dot" option for this same PD and PP, we additionally get the following output:

Output DOT format example:

```
digraph pfile01 {
289589241 [label="Goal"][description="method"][color=red, ordering=out];
289589241 -> 460852438;
289589241 -> 75817673;
460852438 [label="(|get_to#0|␣2␣1␣)"][description="method"][color=red, ordering=
    out];
460852438 -> 2032736387;
460852438 -> -1869216826;
75817673 [label="(|get_to#1|␣2␣0␣)"][description="method"][color=red, ordering=
    out];
75817673 -> -1412872840;
75817673 -> 379927713;
2032736387 [label="(|m_drive_to_ordering_0_Precondition#0|␣0␣1␣2␣)"][description
    ="action"][color=red, ordering=out];
-1869216826 [label="(|drive#1|␣2␣0␣1␣)"][description="action"][color=red,
    ordering=out];
-1412872840 [label="(|m_drive_to_ordering_0_Precondition#0|␣1␣0␣2␣)"][
    description="action"][color=red, ordering=out];
379927713 [label="(|drive#1|␣2␣1␣0␣)"][description="action"][color=red, ordering
    =out];
graph [labelloc="b" labeljust="r" label=<
        <TABLE BORDER="0" CELLBORDER="2" CELLSPACING="0">
        <TR><TD colspan="2">Objects Legend</TD></TR>
<TR><TD>0</TD><TD>c0</TD></TR>
<TR><TD>1</TD><TD>c1</TD></TR>
<TR><TD>2</TD><TD>t0</TD></TR>
</TABLE>>];
}
```

This output then can be converted into the graph in SVG format. For this conversion, we used the online tool Graphviz [13] and got the following diagram (Fig. 3.2).

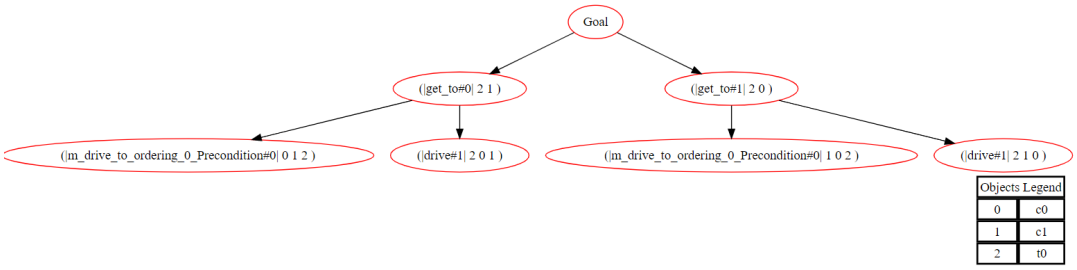


Figure 3.2: Example graph we get after the conversion from the DOT output

4. Encoding

4.1 Encoding of HTN PD and PP into CHC

To encode a HTN planning problem P into a set of CHCs, we follow these steps:

1. We introduce a boolean variable for each ground fact, i.e., for each instantiation of each predicate in our language with constants of appropriate types for the arguments. These variables represent the state of the world at a given stage. If the variable is assigned to true, the ground fact holds in the current state. If the variable is assigned to false, the fact does not hold in the current state. We also use the primed version of the variables to represent the variables in the next state.

2. For each task (both abstract and primitive) $t(x_1, \ldots, x_n)$ we define an uninterpreted predicate $t(i_1, \ldots, i_n, s, s')$ where $i_1, \ldots, i_n$ are variables of sort Integer (each object is mapped to unique Integer indexed from 0), s are the boolean variables introduced in step 1 and s' are the primed version of s. The point behind s and s' is that s is the state before the task or subtask execution and s' is the state right after the execution. Furthermore, in sequence of tasks the s' of task t_1 is equal to the s of task t_2. The intuition behind these uninterpreted predicates is the following: $t(i_1, \ldots, i_n, s, s')$ is true when executing the abstract task t on arguments defined by $i_1, \ldots, i_n$ changes the state from s to s'. i_j for $1 \leq j \leq n$, picks the i_j-th constant from the domain of the type of x_j.

3. For each abstract task t and each of its possible decomposition methods $m : t \rightarrow \langle\ d_1, \ldots, d_m\ \rangle$ (we assume the total order, so the decomposition tasks form a sequence) we introduce a clause of the following form:

$$\begin{aligned} &d_0(args_t, s^{(0)}, s^{(0)}) \ \wedge\ d_1(args_{d1}, s^{(0)}, s^{(1)}) \ \wedge\ \ldots\ \wedge \\ &\quad \wedge\, d_m(args_{d_m}, s^{(m-1)}, s^m) \ \wedge\ Constr_{args} \implies t(args_t, s^{(0)}, s^m) \end{aligned} \tag{4.1}$$

   where $Constr_{args}$ (constraints) ensure that the arguments $args_{d_1}, \ldots, args_{d_m}, args_t$ respect the constraints from the definition of the decomposition method.

   Also, it can happen that the method contains preconditions as well (similarly to the primitive task). As our encoding cannot implicitly work with this, we encode all preconditions of the method into the new child action. This new action inherits the method parameters and preconditions and has no effects. This action is then added as the first subtask of the method itself. From the principle of actions, this is a simple and elegant solution for solving this issue. That is the purpose of the first component $d_0(args_t, s^{(0)}, s^{(0)})$.

4. For each action (primitive task) we introduce a number of clauses, each corresponding to one ground instance of the action.

$$\bigwedge_{p \in pre(a)} p \ \wedge \bigwedge_{e \in eff(a)} e \implies a(args_a, s, s') \tag{4.2}$$

where preconditions are constraints on the current state s and effects are constraints on the next state s'; $args_a$ are numbers enumerating the set of constants of the corresponding argument's type. The set of (boolean) facts in preconditions and effects are given by the concrete constants present in the ground instance of the action.

5. Finally, we introduce a clause defining the initial state s_I and the initial sequence of tasks tn_I .

$$\bigwedge_{s \in s_I} s \ \wedge \bigwedge_{s \notin s_I} \neg s \bigwedge_{t_i \in t_{n_I}} t_i(args_{t_i}, s^{(i-1)}, s^{(i)}) \implies \bot \tag{4.3}$$

This clause encodes the question if the given initial task sequence can be executed from the initial state.

The resulting set of CHCs is satisfiable if and only if there exists no solution to the planning problem. Moreover, if the CHCs problem is unsatisfiable, then the solution for the planning problem can be extracted from the proof of unsatisfiability.

Here may come the question, why should it work, so let's analyze it! We used the famous old Greek motto – "divide et impera". What is that problem we are solving? We ask if a set of abstract tasks is executable. So we divide all abstract tasks into the smallest parts possible – actions. For these actions, we have well-defined preconditions, effects, the state before the action, and the state after the action. Starting with the initial state, the solver checks if there is a satisfiable clause that can be applied to the current state that would bring us closer to the goal state. Supposing the body of implication is true, we can move to the state in the implication head. We continue in this until we have a first evaluation that satisfies the goal, or we end once we conduct none of the actions on our chain that lead to the goal can be executed. This is a tremendous benefit of our approach compared to the others – we can quickly find out the goal is not satisfiable!

4.2 Example task encoded into CHC

In this section we show example PD (3.2) and PP (3.3) encoded into CHC clauses according to steps in previous chapter (in a Prolog format).

1. Boolean variables for all ground instances of predicates:

$$\begin{aligned} &r(c0,c1), r(c1,c0), r(c0,c0), r(c1,c1), \\ &at(t0,c0), at(t0,c1) \end{aligned} \tag{4.4}$$

   Each Boolean variable holds the value of the fact in the domain world. After look at the predicates definition from the Section (3.2), $r(c0,c1)$ holds the information if there is a road from location $c0$ to the location $c1$ and the predicate $at(t0,c0)$ encodes the information whether the vehicle $t0$ is at the location $c0$. The next clauses encode the same information but for other objects.

2. Horn clauses for abstract tasks:

$$\begin{aligned} &get\_to(v,l,\ r(c0,c1)^{(0)}, r(c1,c0)^{(0)}, r(c0,c0)^{(0)}, r(c1,c1)^{(0)}, \\ &\qquad at(t0,c0)^{(0)}, at(t0,c1)^{(0)}, \\ &\qquad r(c0,c1)^{(1)}, r(c1,c0)^{(1)}, r(c0,c0)^{(1)}, r(c1,c1)^{(1)}, \\ &\qquad at(t0,c0)^{(1)}, at(t0,c1)^{(1)}) \\ &:- drive(v,il,l,\ r(c0,c1)^{(0)}, r(c1,c0)^{(0)}, r(c0,c0)^{(0)}, r(c1,c1)^{(0)}, \\ &\qquad at(t0,c0)^{(0)}, at(t0,c1)^{(0)}, \\ &\qquad r(c0,c1)^{(1)}, r(c1,c0)^{(1)}, r(c0,c0)^{(1)}, r(c1,c1)^{(1)}, \\ &\qquad at(t0,c0)^{(1)}, at(t0,c1)^{(1)}) \end{aligned} \tag{4.5}$$

$$\begin{aligned} &get\_to(v,l2,\ r(c0,c1)^{(0)}, r(c1,c0)^{(0)}, r(c0,c0)^{(0)}, r(c1,c1)^{(0)}, \\ &\qquad at(t0,c0)^{(0)}, at(t0,c1)^{(0)}, \\ &\qquad r(c0,c1)^{(2)}, r(c1,c0)^{(2)}, r(c0,c0)^{(2)}, r(c1,c1)^{(2)}, \\ &\qquad at(t0,c0)^{(2)}, at(t0,c1)^{(2)}) \\ &:- get\_to(v,l1,\ r(c0,c1)^{(0)}, r(c1,c0)^{(0)}, r(c0,c0)^{(0)}, r(c1,c1)^{(0)}, \\ &\qquad at(t0,c0)^{(0)}, at(t0,c1)^{(0)}, \\ &\qquad r(c0,c1)^{(1)}, r(c1,c0)^{(1)}, r(c0,c0)^{(1)}, r(c1,c1)^{(1)}, \\ &\qquad at(t0,c0)^{(1)}, at(t0,c1)^{(1)}) \\ &\quad \wedge drive(v,l1,l2,\ r(c0,c1)^{(1)}, r(c1,c0)^{(1)}, r(c0,c0)^{(1)}, r(c1,c1)^{(1)}, \\ &\qquad at(t0,c0)^{(1)}, at(t0,c1)^{(1)}, \\ &\qquad r(c0,c1)^{(2)}, r(c1,c0)^{(2)}, r(c0,c0)^{(2)}, r(c1,c1)^{(2)}, \\ &\qquad at(t0,c0)^{(2)}, at(t0,c1)^{(2)}) \end{aligned} \tag{4.6}$$

   Again, let's analyze one example so we can be sure we understand its meaning in detail. Taking equation (4.6), it is the abstract task *get_to* encoded, specifically *m_drive_to_via_ordering_0* method. We can deduce it from the conjunction of two predicates *get_to* and *drive*. These two together form a

body of this implication where characters ": −" stand for implication mark in a Prolog format. The head of implication is our abstract task *get_to* itself. One may ask why do we have here predicate *get_to* both in the body and head of the implication. The question is rather simple, their meaning there is a bit different, and the clause is just an encoded method with subtasks from the definition. You have probably already noticed different variables. While the head *get_to* predicate has parameters v and $l2$, in the body *get_to* we have v and $l1$. These object variables are mapped to unique Integer numbers, which we describe later. After predicate parameters, two different sets of ground predicate variables follow. The first one holds the state before the task, the second holds the state after the task execution. In the head, the index of a first set is always 0, and the index of the second set is the number of subtasks in the body – our example has two subtasks, so the second index is 2. In the body predicates, we start with the indexes 0 and 1 for the first subtask, the second starts where the previous ended, so 1 and continues with 2 and so on. You can again notice the motivation there. The output state of one subtask is equal to the input state of the next, which means that also these sets are the same. Overall, supposing all predicates from the body are true, the abstract task from the head of implication can be executed.

3. Horn clauses for primitive tasks:

$$\begin{aligned} &drive(2,0,0,\, r(c0,c1)^{(0)}, r(c1,c0)^{(0)}, r(c0,c0)^{(0)}, r(c1,c1)^{(0)},\\ &\qquad at(t0,c0)^{(0)}, at(t0,c1)^{(0)},\\ &\qquad r(c0,c1)^{(1)}, r(c1,c0)^{(1)}, r(c0,c0)^{(1)}, r(c1,c1)^{(1)},\\ &\qquad at(t0,c0)^{(1)}, at(t0,c1)^{(1)})\\ &:- at(t0,c0)^{(0)} \wedge r(c0,c0)^{(0)} \wedge \neg at(t0,c0)^{(1)} \wedge at(t0,c0)^{(1)}\\ &\qquad \wedge r(c0,c1)^{(0)} = r(c0,c1)^{(1)} \wedge r(c1,c0)^{(0)} = r(c1,c0)^{(1)}\\ &\qquad \wedge r(c0,c0)^{(0)} = r(c0,c0)^{(1)} \wedge r(c1,c1)^{(0)} = r(c1,c1)^{(1)}\\ &\qquad \wedge at(t0,c1)^{(0)} = at(t0,c1)^{(1)} \end{aligned} \tag{4.7}$$

$$\begin{aligned} &drive(2,0,1,\, r(c0,c1)^{(0)}, r(c1,c0)^{(0)}, r(c0,c0)^{(0)}, r(c1,c1)^{(0)},\\ &\qquad at(t0,c0)^{(0)}, at(t0,c1)^{(0)},\\ &\qquad r(c0,c1)^{(1)}, r(c1,c0)^{(1)}, r(c0,c0)^{(1)}, r(c1,c1)^{(1)},\\ &\qquad at(t0,c0)^{(1)}, at(t0,c1)^{(1)})\\ &:- at(t0,c0)^{(0)} \wedge r(c0,c1)^{(0)} \wedge \neg at(t0,c0)^{(1)} \wedge at(t0,c1)^{(1)}\\ &\qquad \wedge r(c0,c1)^{(0)} = r(c0,c1)^{(1)} \wedge r(c1,c0)^{(0)} = r(c1,c0)^{(1)}\\ &\qquad \wedge r(c0,c0)^{(0)} = r(c0,c0)^{(1)} \wedge r(c1,c1)^{(0)} = r(c1,c1)^{(1)} \end{aligned} \tag{4.8}$$

$$
\begin{aligned}
&drive(2,1,0,\, r(c0,c1)^{(0)}, r(c1,c0)^{(0)}, r(c0,c0)^{(0)}, r(c1,c1)^{(0)},\\
&\qquad at(t0,c0)^{(0)}, at(t0,c1)^{(0)},\\
&\qquad r(c0,c1)^{(1)}, r(c1,c0)^{(1)}, r(c0,c0)^{(1)}, r(c1,c1)^{(1)},\\
&\qquad at(t0,c0)^{(1)}, at(t0,c1)^{(1)})\\
&:- at(t0,c1)^{(0)} \wedge r(c1,c0)^{(0)} \wedge \neg at(t0,c1)^{(1)} \wedge at(t0,c0)^{(1)}\\
&\qquad \wedge r(c0,c1)^{(0)} = r(c0,c1)^{(1)} \wedge r(c1,c0)^{(0)} = r(c1,c0)^{(1)}\\
&\qquad \wedge r(c0,c0)^{(0)} = r(c0,c0)^{(1)} \wedge r(c1,c1)^{(0)} = r(c1,c1)^{(1)}
\end{aligned}
\tag{4.9}
$$

$$
\begin{aligned}
&drive(2,1,1,\, r(c0,c1)^{(0)}, r(c1,c0)^{(0)}, r(c0,c0)^{(0)}, r(c1,c1)^{(0)},\\
&\qquad at(t0,c0)^{(0)}, at(t0,c1)^{(0)},\\
&\qquad r(c0,c1)^{(1)}, r(c1,c0)^{(1)}, r(c0,c0)^{(1)}, r(c1,c1)^{(1)},\\
&\qquad at(t0,c0)^{(1)}, at(t0,c1)^{(1)})\\
&:- at(t0,c1)^{(0)} \wedge r(c1,c1)^{(0)} \wedge \neg at(t0,c1)^{(1)} \wedge at(t0,c1)^{(1)}\\
&\qquad \wedge r(c0,c1)^{(0)} = r(c0,c1)^{(1)} \wedge r(c1,c0)^{(0)} = r(c1,c0)^{(1)}\\
&\qquad \wedge r(c0,c0)^{(0)} = r(c0,c0)^{(1)} \wedge r(c1,c1)^{(0)} = r(c1,c1)^{(1)}\\
&\qquad \wedge at(t0,c0)^{(0)} = at(t0,c0)^{(1)}
\end{aligned}
\tag{4.10}
$$

Clauses for primitive tasks, as the name implies, are simple compared to abstract tasks. The head predicate, which is the name of primitive task with its parameters and again clauses for the state before and after the task execution, has only indexes 0 and 1 as they represent just one simple step. In the body, we have conjunctions of all preconditions (predicates with upper index 0), effects (predicates with upper index 1), and clauses representing that the predicate is not affected by the task. This is expressed in the way that predicate before the task execution has the same value as the predicate after the task is performed. For example, $r(c0,c1)^{(0)} = r(c0,c1)^{(1)}$ says that if there was a road from the location $c0$ to the location $c1$ before the primitive task execution, the same is true after the primitive task is executed. Either it is in both cases true or in both cases false.

4. Clause for the initial state and initial task:

$$
\begin{aligned}
\bot \;:-\; &at(t0,c0)^{(0)} \wedge \neg at(t0,c1)^{(0)}\\
&\wedge r(c0,c1)^{(0)} \wedge r(c1,c0)^{(0)} \wedge \neg r(c0,c0)^{(0)} \wedge \neg r(c1,c1)^{(0)}\\
&\wedge get\_to(0,1,r(c0,c1)^{(0)}, r(c1,c0)^{(0)}, at(t0,c0)^{(0)}, at(t0,c1)^{(0)},\\
&\quad r(c0,c1)^{(1)}, r(c1,c0)^{(1)}, at(t0,c0)^{(1)}, at(t0,c1)^{(1)})
\end{aligned}
\tag{4.11}
$$

Finally, encoding of initial state and goal tasks is pretty straightforward. The head of implication holds a contradiction, and the body contains conjunction of all initial predicates listed in the *init* block of the problem file as

true, negations of all other ground variables predicates, and all goal tasks predicates. Indexes in the goal tasks share common logic as in the subtasks from abstract tasks. 0 and 1, 1 and 2, and so forth.

4.3 Encoding preparation

Correct data in a suitable format is a significant precondition for each transformation process, and this is not an exception. Once PD and PP are parsed, we possess data in the raw format in which it was given. However, for our purpose, we need to add some minor modifications and additions. Technically, this is a smaller substep, but logically it is an inseparable part of the whole procedure. As these functions are not numerous, we included them in this chapter. We call it Data/Problem enrichment, and it consists of few primary steps.

The main tasks of Problem enrichment are following:

- In a PD there are type definitions in form of *type* - *basetype*. However, There can be multiple types inheritance, so we have to map subtypes for each type. For example, there is type A inherited by type B, and type B is inherited by type C. Supposing we have object o of type C, it means that it is also of type B and C. Without this knowledge, in case there is a method m that takes one parameter of type A, our object would not be assigned as a good fit.
- Create variables for all ground instances of a predicate. This is done by calculating all permutations of each predicate with acceptable objects.
- Generate mapping for each object to a unique Integer number, which is later used in the encoding. The purpose behind this is that Spacer engine from the Z3 theorem prover does not support String variables. In the end, we map these numbers back to object names.
- Create a set of predicates for each state in abstract tasks so we can store the state between steps.
- Set initial values where needed (for example, predicates that are not in the initial state must be set to false).
- Encode method preconditions in each method into a new action that is inserted as a first method subtask. This is also done in case the method does not have any precondition. The purpose behind this is that we include a method name into the name of this action so we can later recognize which concrete method was used to accomplish the abstract task and use this information in plan reconstruction.
- Order method subtasks according to explicit ordering if present.

4.4 Technical perspective of encoding

After Problem enrichment adjustments are made, everything is ready for the encoding steps from chapter 4.1. There comes our encoder, which executes the steps presented. Each CHC is encoded in format for the Z3 theorem prover. We use the encoding presented with slight changes. As the order of the subtasks cannot be forced, we used a minor workaround there. Each clause is created multiple times – with every possible order index it can have where the count is the maximal number of subtasks in a task, no matter if it is a goal task or some abstract task. The purpose for this is to include the index of the order to the clause itself, so it is visible in the final answer we get from the solver, thus the order of execution is clear. This is not a satisfactory solution, but currently, it was the easiest to implement without change in encoding. After all clauses are encoded, we query if the clause for the goal state is satisfiable or not. In case the answer is positive, we construct a transition graph of such answer, construct a plan using the depth-first search and return it in the form of standard output. In case of negative answer, we state that a solution to the given problem within a given domain does not exist.

5. Developer documentation

The program consists of several sections. The main parts are Parser, Data enricher, Z3 encoder, and Answer extractor (There comes handy flow diagram from the chapter 2.1). We will decompose each part in the following sections so the orientation in the code, its understanding, and possible future alterations are smooth. The whole program is composed as a Maven project and uses Open-JDK 15.0.1. All the following parts are located in the package `com.kazimirm.transitionSystemBasedHtnSolver` so whenever we mention some package in this chapter, it is a subpackage of this package. The main method is located in this package in the file `TransitionSystemBasedHtnSolver.java`.

5.1 Parser

The parser, as mentioned in the Section 3.2, is implemented using JavaCC [6] where the hearth of it is a grammar including the rules in the file `parser.jj` from the package `parser`. This package then additionally contains other classes that were generated by JavaCC.

The benefit of this solution is that in case of any change regarding the grammar, regenerating new java parser files is easy. Supposing that on Your machine is JavaCC and Java correctly installed and path set, just open a command prompt in a location of the parser file and execute the following command: "`javacc parser.jj`". All the auxiliary files will be updated by the changes done in the grammar file. There is also a need to rebuild the project again (see Section 6.2).

The parser can parse both domain and problem files using respective methods – `parseDomain()` and `parseProblem()` where the return value of these methods is a respective object. The Figure 5.1 shows the class diagram of these data classes.

All input data classes are located in the package `hddlObjects`. Domain and problem extend class `HtnInput` with just one variable of enum type `InputType` that currently has two values – `DOMAIN` and `PROBLEM` for future extensions and to simplify how we can work with such input objects.

We will not go through each class as it is not that interesting. Each class just copies the structure of its section in the input file where we use `String` for simple concrete variables or create subclass to represent some bigger part. For example, let's take the list of objects from the `Problem` class – `List<Argument> objects;`. It is the list that contains objects of type `Argument`.

`Argument` is a simple class with two `String` variables, each has a getter and a setter method. Additionally, there is `equals` method, so we can compare two objects of type `Argument` if they are the same.

Other classes use the same principle. Moreover, there can be found slightly amended `get`, `print`, `equals` or `clone` methods. For storing collections where we want to preserve the order of insert, we use `ArrayList<T>` or `LinkedHashMap<T, T>` objects. Where this is not needed, we use just simple `HashMap<T, T>` objects or `ArrayList<T>` as well. Some properties are not set

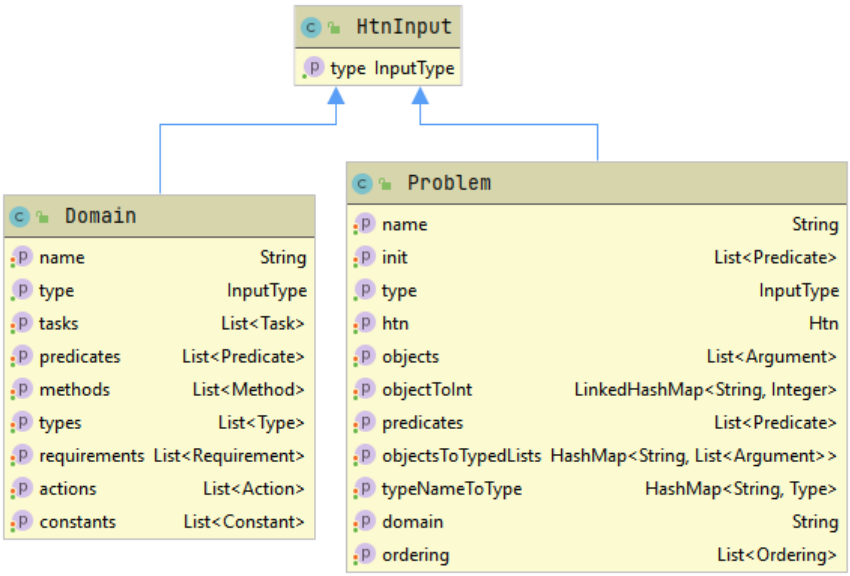


Figure 5.1: Class representation of parsed input

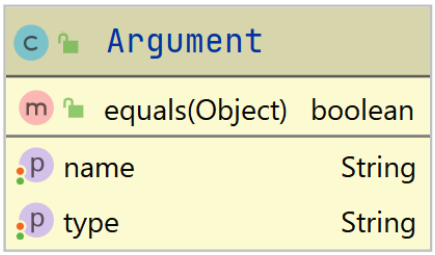


Figure 5.2: Class Argument diagram

during the parsing but in the following part called Data enrichment.

5.2 Data enricher

The purpose of data enricher should be clear by now. Simply said, some minor adjustments must be done to our raw data once they are parsed before we can start the encoding. All the work is ensured by class `ProblemEnricher` in the package `dataEnricher` which at the beginning takes our `Domain` and `Problem` objects.

```
ProblemEnricher enricher = new ProblemEnricher(domain, problem);
```

We create an instance of `ProblemEnricher` because it just amends the input domain and problem parameters and has some internal properties as well. The compact view of `ProblemEnricher` class can be seen in Figure 5.3.

All the Main tasks are then done by these few methods:

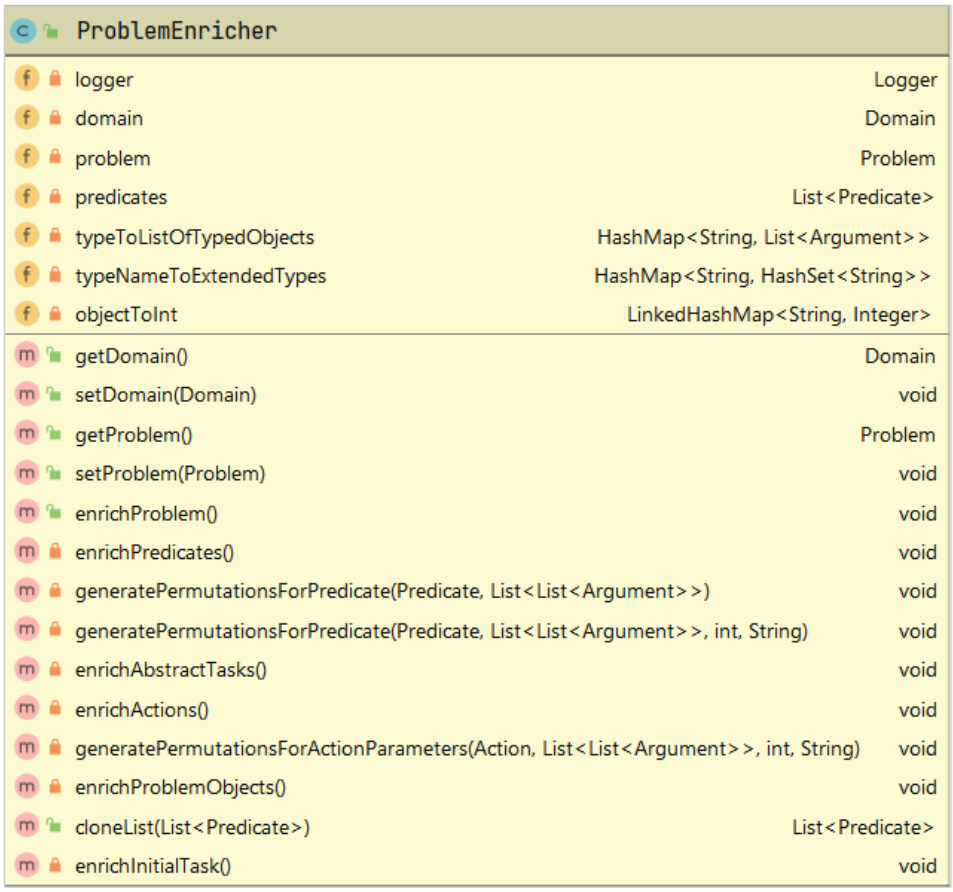


Figure 5.3: Class ProblemEnricher diagram

- `enrichProblemObjects()`
  Creates unique mapping for each object into an `Integer` number by filling the `objectToInt` because the Z3 Spacer engine does not support `String` variables. This functionality could be solved in the parser, but we wanted to separate the parser from other functionalities and other logic.

- `enrichPredicates()`
  This method creates variables for all ground instances of predicates (calculates the permutations using `generatePermutationsForPredicate` overloaded methods based on recursion and fills them into the list `predicates`). Also, it maps all subtypes for each type to support multiple inheritance of data types (this is the purpose and stage when `typeNameToExtendedTypes` collection is filled). For each type, we fill the `typeToListOfTypedObjects` collection where we insert the problem objects in case they are of that type or subtype.

- `enrichAbstractTasks()`
  Firstly, we handle encoding method preconditions into a new action that is inserted before all the subtasks. The reason behind this is that methods with preconditions have unique handling because our encoding does not support that. We create an action for each method that has some preconditions. This action has the same parameters and preconditions as the

method, no effects, and it will be the first subtask of the method.

Secondly, we add sets of predicates with correct indexes to each subtask and the method task itself. In the end, we check the order of subtasks in case it was defined in some different order (that is unnecessary in the current state but serves for the future extension to support not only totally-ordered input).

- `enrichActions()`
  Creates permutations of objects for every action argument and adds indices to precondition and effects predicates.

- `enrichInitialTask()`
  Same purpose and principle as in the `enrichAbstractTasks()` but for goal subtasks. Also, we don't expect the initial task to have preconditions, so this part is skipped.

At the end, we also set `typeToListOfTypedObjects`, `objectToInt` and `predicates` variables into the problem instance as these collections will be later needed. Once our PD and PP are updated with these changes, nothing stands in the way of the encoding phase.

5.3 Z3 encoder

The encoding phase is done via class `Z3Encoder` from the package `encoder`. As the Z3 provides Java API it makes things easier. Again, there is a simple workflow to encode the whole problem and get the answer in the form of a tree represented using arrays, but before we start, we set three options at the beginning.

```
1  public void encodeToZ3ExpressionsAndGetResult() {
2          fix = ctx.mkFixedpoint();
3
4          // few settings for Z3 encoder before we start
5          Params params = ctx.mkParams();
6          params.add("xform.slice", false);
7          params.add("xform.inline_linear", false);
8          params.add("xform.inline_eager", false);
9          fix.setParameters(params);
10
11         encodeVariables();
12         encodeFunctionSignatures();
13         encodeActions();
14         encodeMethods();
15         encodeHtnAndInit();
16     }
```

Parameters at the lines 6, 7 and 8 ensure we will be able to reconstruct the plan from the Z3 FixedPoint engine (Spacer) proof should the CHC system of clauses be unsatisfiable. Also, on the `Z3Encoder` class instantiation, we create a new Z3 context (`Context ctx = new Context()`) under which all Z3 objects are attached.

- `encodeVariables()`
  Creates boolean variables for every ground variable. Number of each variable instances must be equal to max + 1 number of maximum subtasks in a method. Then we will create list of lists which will be used in tasks/methods/... encoding. Variables are created via method

  `ctx.mkBoolConst(String var1)`.

- `encodeFunctionSignatures()`
  For each task and action we need to declare function signature with all its parameters. This is done via method

  `ctx.mkFuncDecl(String var1, Sort[] var2, R var3);` where `var1` is name of the function, `var2` is array of `Sort` objects (object variables from clauses are of type `IntSort` created by `ctx.mkIntSort()` and boolean variables are of type `BoolSort` created by `ctx.mkBoolSort()`)

- `encodeActions()`
  Encoding actions into Z3 rules and adding them to the FixedPoint. Encoding of actions is a bit longer, and expression is built gradually. Firstly, we add the preconditions. In case the precondition is negative, the negation is created using `ctx.mkNot(Expr<BoolSort> var1)`. All these preconditions are added to the list of `BoolExpr` type. Let's mark it as a list `A`. We apply the same principle to encode the effects and add them to the same list `A`. For predicates that are not changed by the action, we encode that the state should not be changed. This is done by making a `BoolExpr` expression of the equality via `ctx.mkEq(Expr<R> var1, Expr<R> var2);` where var1 is state predicate before the task and var2 is state predicate after the task execution. Also, these expressions are added to the list `A`. Then we create a second list which we can mark as a list `B`. We insert all action parameter variables (objects) into that list, predicates variables before and after the action, and apply a suitable function using `ctx.mkApp(FuncDecl<R> var1, Expr<?>... var2)`. In the end, we create an implication of the rule from our list `A` and `B` via

  `ctx.mkImplies(Expr<BoolSort> var1, Expr<BoolSort> var2)` Also, before we can add it to the FixedPoint, we have to apply quantification by converting our `Expr` into the `Quantifier` object using few methods as follows:

```
1 Expr expr = ctx.mkImplies(ruleA, ruleB);
2 allExpressions.add(expr); // for debugging purposes
3 Quantifier quantifier = ctx.mkForall(ruleBParams.toArray(new
     Expr[0]), expr, 0, null, null, null, null);
4 Symbol symbol = ctx.mkSymbol(a.getName() + HASH + i +
     permutation.toString());
5 fix.addRule(quantifier, symbol);
```

  `Symbol` serves as the name of the expression so we construct it using unique combination of action name, character '#' and permutation of objects.

- `encodeMethods()`

For each method - the implication rule is created in a similar manner as in the case of actions. If all method subtasks are satisfiable, the method task is satisfiable. Firstly we create the expression of conjunctions for each subtask, then using this expression implication for the task is created and added to FixedPoint. Additionally, there can occur constraints in the methods. Constraint means that if a method has more parameters of the same type, the objects cannot equal, and we handle it by creating the disequality:

```
IntExpr param1 = intExpressions.get(c.getParam1().getName());
IntExpr param2 = intExpressions.get(c.getParam2().getName());
Expr eq = ctx.mkNot(ctx.mkEq(param1, param2));
```

- `encodeHtnAndInit()`
  Encoding of the HTN and initial state into Z3 rules and adding it to FixedPoint. We encode the initial state by a process similar to the process of encoding methods and create a new empty expression to serve as the goal state. Once this is done, everything is encoded, and we query the FixedPoint whether the goal state is satisfiable from the initial state and get the answer in the form of expression via `Expr<?> getAnswer()`. The extraction of the answer from this `Expr` object is performed in the last part – Answer extractor.

5.4 Answer extractor

The outcome of the Z3 encoder that is interesting for us consists of two parts. One would be a simple boolean value if we had success or not. In case we found the solution, we need to extract it and construct a plan from the second part - the expression. We instantiate the class `answerExtractor.AnswerExctractor` using the `encoder` object from the previous step. The Z3 expression is in the form of an array. Each array has objects that represent nodes. Each node is then again array with other nodes. As working with this representation would be difficult, we transform it to our graph representation located in the package `answerExtractor.graphRepresentation` based on an adjacent map. On this instance, we call `Graph getGraphFromAnswer()` that returns us our graph representation of the Z3 expression. We construct the graph using a breath-first search algorithm in the `Graph` class – `void createGraph()`. We traverse each level and take the relevant nodes - nodes of type `Z3_OP_PR_HYPER_RESOLVE`. We can distinguish primitive tasks from abstract ones according to the number of children elements. If the number of children is bigger than 2, the node belongs to the abstract task. Once we have our graph created, in order to get the plan, we just call a `String graph.getStandardOutput()` that performs the depth-first search again but on our graph representation starting in the root node that collects nodes and puts into the `StringBuilder` in the standard output format, merges together and returns the plan in a `String` format (Firstly, there is a check if the plan exists and if does not, we just print the negative answer to the console).

6. User documentation

6.1 Restrictions and dependencies

The usage of solver is pretty simple without numerous demands. It comes in a form of executable jar so it runs using commands `java jar jarname`. The only precondition we expect is that user has installed Java JDK 15, Maven and Z3 theorem prover [14]. Also make sure `JAVA_HOME` variable is correctly set as well as environment paths on used machine. Presented build steps were tested on the Windows platform but the process is analogical on Unix as well.

These dependencies can be easily checked. For Java, open command prompt and type `'java -version'`. The response should be similar to example below (in our case we used OpenJDK).

```
C:\Users\testing>java -version
openjdk version "15.0.1" 2020-10-20
OpenJDK Runtime Environment (build 15.0.1+9-18)
OpenJDK 64-Bit Server VM (build 15.0.1+9-18, mixed mode, sharing)
```

Maven installation can be checked by command `'mvn -version'`. Again, the response should look like in our example:

```
C:\Users\testing>mvn -version
Apache Maven 3.8.1 (05c21c65bdfed0f71a2f2ada8b84da59348c4c5d)
Maven home: C:\Program Files (x86)\Maven\apache-maven-3.8.1-bin\apache-maven-3.8.1\bin\..
Java version: 15.0.1, vendor: Oracle Corporation, runtime: C:\Users\cekvi\.jdks\openjdk-15.0.1
Default locale: en_US, platform encoding: Cp1250
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

Z3 theorem prover installation can be checked by command `'Z3 -version'`. The response should look like this:

```
C:\Users\testing> Z3 -version
Z3 version 4.8.10 - 64 bit
```

6.2 Build

Firstly, project repository should be downloaded. Once the files are unzipped at a desired location, open the command prompt and execute these commands:

```
cd $PROJECT_ROOT_DIRECTORY
mvn clean install
```

Part of the build process is running some tests. This phase can be skipped by adding `'-DskipTests'` option at the end of second line. This will speed up the build process. Once the build is done, jar file is generated into the directory `target`.

6.3 How to run

The program jar file takes these parameters split by space:

- path to file of the HTN domain in HDDL format as the first parameter.

- path to the file of HTN problem in HDDL format (relevant to the domain from previous parameter) as the second parameter.
- Optionally, if a third parameter `'-dot'` is present, in addition to the standard output, also output in a form of graph in the DOT format is printed.

Solver uses few helpful prints so each part of the job can be evaluated. Example:

```
C:\Users\testing\transition-system-based-solver\target>java -jar
    Transition-system-based_HTN_planning_solver-1.0-SNAPSHOT.jar C:\Users\testing\domain.hddl
    C:\Users\testing\problem.hddl
Solver has started...
Parsing phase initialized...2021-07-13T23:51:30.992593600
Parsing phase ended...2021-07-13T23:51:31.022094900
Enrichment phase initialized...2021-07-13T23:51:31.022593
Enrichment phase ended...2021-07-13T23:51:31.535094200
Encoding phase initialized...2021-07-13T23:51:31.535594600
Encoding phase ended...2021-07-13T23:51:32.157129
Extracting the result...2021-07-13T23:51:32.157600200
Result extracted...2021-07-13T23:51:32.237087700
Printing the result:
==>
14 drive truck_0 city_loc_2 city_loc_1
16 pick_up truck_0 city_loc_1 package_0 capacity_0 capacity_1
18 drive truck_0 city_loc_1 city_loc_0
20 drop truck_0 city_loc_0 package_0 capacity_0 capacity_1
22 drive truck_0 city_loc_0 city_loc_1
24 pick_up truck_0 city_loc_1 package_1 capacity_0 capacity_1
26 drive truck_0 city_loc_1 city_loc_2
28 drop truck_0 city_loc_2 package_1 capacity_0 capacity_1
root 1 2
1 deliver package_0 city_loc_0 -> m_deliver_ordering_0 4 5 6 7
4 get_to truck_0 city_loc_1 -> m_drive_to_ordering_0 14
5 load truck_0 city_loc_1 package_0 -> m_load_ordering_0 16
6 get_to truck_0 city_loc_0 -> m_drive_to_ordering_0 18
7 unload truck_0 city_loc_0 package_0 -> m_unload_ordering_0 20
2 deliver package_1 city_loc_2 -> m_deliver_ordering_0 9 10 11 12
9 get_to truck_0 city_loc_1 -> m_drive_to_ordering_0 22
10 load truck_0 city_loc_1 package_1 -> m_load_ordering_0 24
11 get_to truck_0 city_loc_2 -> m_drive_to_ordering_0 26
12 unload truck_0 city_loc_2 package_1 -> m_unload_ordering_0 28
<==

Execution took '1.2811759' seconds
```

In case of success, found plan is print in the standard format[12].

7. Evaluation

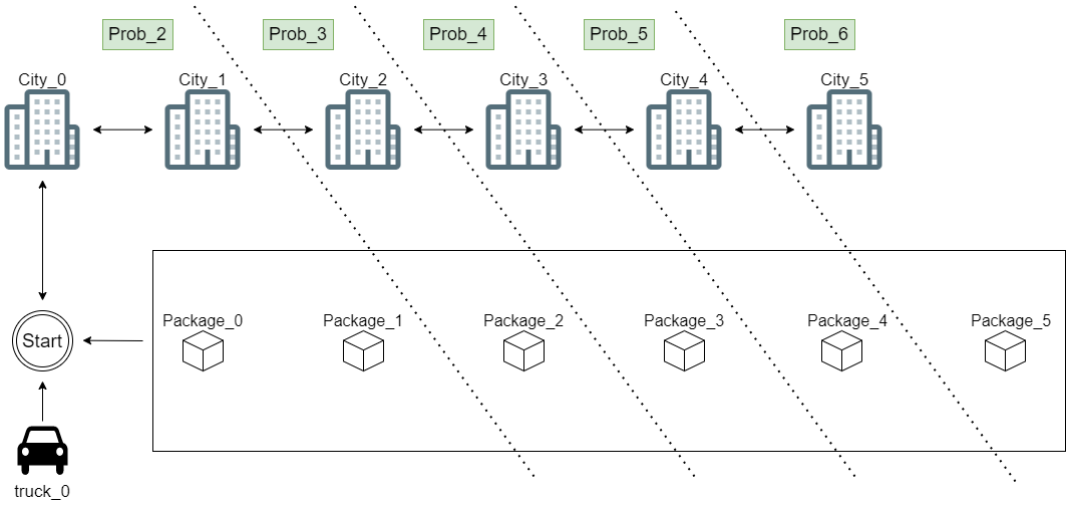
7.1 Basic tests dataset

The dataset used in our basic comparison consisted of four totally-ordered tasks from the 2020 IPC for Hierarchical Planning. We'll come through every domain and problem in it to see their size and possible purpose.

- **rover01** domain contains 7 simple object types without inheritance, 26 predicates, 9 abstract tasks, 13 methods[1] and 11 actions. The problem in it has 13 objects, and the goal is to execute 3 abstract tasks.
- **satellite01** domain contains 5 object types without inheritance, 8 predicates, 3 abstract tasks, 8 methods, and 5 actions. The problem in it has 6 objects, and the goal has just one abstract task.
- **um-translog01** domain is the most enormous with 163 object types with multiple type inheritance, 34 predicates, 21 abstract tasks, 52 methods, and 51 actions. The problem in it has 15 objects, and the goal has just one abstract task.
- **transport01** domain has 6 object types with inheritance, 5 predicates, 4 abstract tasks, 6 methods, and 4 actions. Its problem has 8 objects and the goal is the execution of 2 abstract tasks.

7.2 Advanced level dataset

We wanted to see how our solver behaves when the problem grows. We chose the "transport01" domain and created few problems from simple to more elaborate. The principle of the extension was adding objects and expanding the goal within the fixed domain in the following matter:



[1]Mind that the method is a concrete implementation of an abstract task thus one abstract task can have multiple methods.

According to the problem name - `"Prob_i"` has `i` cities, `i` packages, and 1 vehicle located at the start location. There is a road between every two adjacent cities, and the capacity of the car is 1 package. The goal is to move every `package_i` to the `City_i` so the vehicle must take the first package to the first city, then come back, take the second package to the second city, and so on.

7.3 Comparison

In order to compare our solver with competitive ones, we took solvers from the 2020 IPC for Hierarchical Planning that were available. We had numerous problems because some of them were without any documentation, and some of them had few issues.

Finally, we were able to launch two of them - Lilotane[15] written in C++ and HyperTensioN[16] written in Python. Their results can be found at IPC 2020 competition status page[11]. We tested each solver against the basic tests dataset(7.1) on our test environment and got the results beneath.

Solver	rover01	satellite01	um-translog01	transport01
Lilotane	0.11s	0.06s	0.28s	0.06s
HyperTensioN	0.32s	0.03s	0.37s	0.12s
Our	4.8s	1.5s	50s	1.3s

Table 7.1: table with execution times of basic dataset (7.1)

As it is seen, our solver is significantly slower. However, this is the result we expected for multiple reasons.

Firstly, our solver is written in Java, so the code must be first interpreted during run-time. In the case of execution speed, C++ is faster than Java, but Java is faster than Python, yet we use a different approach than compared ones.

Secondly, our aim was to conquer in problems that are complicated and hoped for better results in our second test. So we tested all three candidates against our advanced level dataset(7.2) and got another results.

Solver	Prob_2	Prob_3	Prob_4	Prob_5	Prob_6
Lilotane	0.07s	0.09s	0.14s	0.2s	0.3s
HyperTensioN	0.01s	0.02s	0.05s	0.13s	0.9s
Our	1.3s	6.1s	17s	69s	350s

Table 7.2: table with execution times of level dataset (7.2)

While in the case of competitive solvers, the time of execution rose just slightly, in our case, the time of execution grew almost exponentially to the problem size. The problem, in this case, is that we extend the problem by objects of the same type. It means that with every addition we increase the number of encoded clauses dramatically. The reason is rather simple - not only do we add one additional object, so we increase the number of permutations for all predicates,

we also extend the goal by an additional abstract task, which is another non-trivial addition to a number of encoded clauses. Here comes the technical debt of our solution for actions order in methods (4.4). Every action is encoded to a maximal number of actions the method can have. This, with the combination of growing permutations for action parameters, causes this effect.

Nevertheless, that doesn't mean our solver is useless. One of the benefits our solver has is that we can decide whether the problem does have a solution or does not. When we presented some undoable problems to mentioned solvers, they didn't end - they were in an infinite cycle of finding a plan, yet there is none. Of course, it comes with the price of time. What's more, the execution is sequential, so should there be a problem in the later subtask, it would take some time to come to this conclusion.

In addition, the approach itself has good advantages, and there are multiple bottlenecks that can be improved. In case we could find a way how to encode actions order better, the results would improve extremely. Moreover, we could add some heuristics so we do not create clauses that we know are impossible. Last but not least, the third test example from the basic dataset `um-translog01` (7.1) shows us that multiple type inheritance is something our solver has problems with.

Conclusion

In this work, we have devoted ourselves to the subset of planning problems – Hierarchical Task Network problems. After doing research, we found out that the most commonly used solvers use the incremental SAT approach. We considered the sequential building of fixed-size queries that this approach includes as the main disadvantage, so we were looking for ways to avoid it.

We designed an encoding into the reachability problem that holds all the necessary information. A problem in such encoding is a system of Constrained Horn Clauses, and if that system is unsatisfiable, we can extract the plan from the proof of unsatisfiability. So not only do we get the plan in case the task has some solution, but we can also determine if the problem has no solution.

Our research then continued by technical aspects – how to make it work. We checked all the possibilities and came with a clear strategy that led us to the architecture composed of several parts. The emphasis was taken on the speed of implementation, simplicity, and best practices. We were step by step building the application so we could test our attitude and compare its effectiveness with some solvers based on the common approach.

Firstly we created a parser to get the input and store it in a reasonable internal representation. Secondly, we needed to make few minor amendments so the encoding would be possible. Then, we implemented encoding followed by computation by the Z3 theorem prover [7]. Lastly, we needed to develop an algorithm to extract the plan from the answer provided by Z3.

Once all the parts were implemented, we created a test strategy on how to make some benchmarks giving us the feedback. Not only we tested it against the set of tests used on the 2020 International Planning Competition [11], we also created our own test set to check how the solvers behave with the growing problems.

Yet it is true that our solver could not cope with the competitive ones in the matter of speed, its benefit was the ability to detect that the plan does not exist. Both competitive solvers did not have this ability, and the computation was endless. This makes our solver be a great fit in the area where we do not know if any plan exists and requirements on the speed are not that high.

What's more, we identified numerous bottlenecks in our approach. I would point out especially the current solution how we ensure the right order of subtasks. We believe that solving them would improve the time of execution tremendously.

To sum up, we verified that HTN solver based on translating the planning problem to a reachability problem works and has its potential. At the moment, we have a stable working solver and few ideas on how to enhance its effectiveness and overall results.

Bibliography

[1] Wikipedia, 17.7.2021 Automated planning and scheduling, available at *https://en.wikipedia.org/wiki/Automated_planning_and_scheduling*

[2] Holler, et al., 2019a Daniel Holler, Gregor Behnke, Pascal Bercher, Susanne Biundo, Humbert Fiorino, Damien Pellier, and Ron Alford. *HDDL – A language to describe hierarchical planning problems.* In Proc. of the 2nd ICAPS Workshop on Hierarchical Planning, 2019, available at https://www.researchgate.net/publication/334448600_HDDL_-A_Language_to_Describe_Hierarchical_Planning_Problems

[3] Bjørner N., Gurfinkel A., McMillan K., Rybalchenko A. (2015) Horn Clause Solvers for Program Verification. In: Beklemishev L., Blass A., Dershowitz N., Finkbeiner B., Schulte W. (eds) Fields of Logic and Computation II. Lecture Notes in Computer Science, vol 9300. Springer, Cham., available at *https://doi.org/10.1007/978-3-319-23534-9_2*

[4] A. Gurfinkel and N. Bjørner, "The Science, Art, and Magic of Constrained Horn Clauses", 2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2019, pp. 6-10, doi: 10.1109/SYNASC49474.2019.00010., available at *https://ieeexplore.ieee.org/abstract/document/9049884*

[5] DanaNau YueCao AmnonLotem Hector Muftoz-Avila, "SHOP: Simple Hierarchical Ordered Planner", 1999 epartment of Computer Science, and Institute for Systems Research University of Maryland, College Pa, available at *https://www.ijcai.org/Proceedings/99-2/Papers/043.pdf*

[6] JavaCC community Official page, 2021 - JavaCC community *https://javacc.github.io/javacc/*

[7] de Moura L., Bjørner N. (2008) Z3: An Efficient SMT Solver. In: Ramakrishnan C.R., Rehof J. (eds) Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2008. Lecture Notes in Computer Science, vol 4963. Springer, Berlin, Heidelberg., available at *https://doi.org/10.1007/978-3-540-78800-3_24*

[8] Komuravelli, A., Gurfinkel, A. & Chaki, S. SMT-based model checking for recursive programs. Form Methods Syst Des 48, 175–205 (2016), available at *https://doi.org/10.1007/s10703-016-0249-4*

[9] Rummer, P., Competition Report: CHC-COMP-20, 2020 available at *https://arxiv.org/pdf/2008.02939.pdf*

[10] ©Microsoft, 2021 - Z3 Spacer engine tutorial *https://rise4fun.com/Z3/tutorial/guide*

[11] International Planning Competition 2020, official page, 2020 Universität Freiburg *http://gki.informatik.uni-freiburg.de/competition/*

[12] International Planning Competition, official page, 2020 - IPC standard format definition *https://gki.informatik.uni-freiburg.de/ipc2020/format.pdf*

[13] Michael Daines, Github, 2021 - Graphviz online tool available at *https://dreampuf.github.io/GraphvizOnline*

[14] Z3Prover, Github, 2021 - Github page with Z3 theorem prvoer releases *https://github.com/Z3Prover/z3/releases*

[15] Schreiber D., 2021 - Lilotane: A Lifted SAT-based Approach to Hierarchical Planning. In Journal of Artificial Intelligence Research (JAIR) 2021 (70), pp. 1117-1181. *https://github.com/domschrei/lilotane*

[16] Maurício Cecílio Magnaguagno, Felipe Meneguzzi, Lavindra de Silva - HyperTensioN: A three-stage compiler for planning *https://github.com/Maumagnaguagno/HyperTensioN*

List of Figures

List of Tables